Downloaded from orbit.dtu.dk on: May 05, 2023

DTU Library

Algorithms and programs for consequence diagram and fault tree construction

Hollo, E.; Taylor, J.R.

Publication date:
1976

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Hollo, E., & Taylor, J. R. (1976). *Algorithms and programs for consequence diagram and fault tree construction*. Risø National Laboratory. Risø-M No. 1907

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

A. E. K. Risø

Risø - M - 1907

Risø - M - 1907

Title and author(s)

Algorithms and programs for consequence diagram and fault tree construction.

E. Hollo* and J.R. Taylor

Date December 1976

Department or group

Electronics

Group's own registration number(s)

R-1-77

pages + tables + illustrations

Abstract

Algorithms and programs for consequence diagram and sequential fault tree construction are presented which are intended for reliability and disturbance analysis of large systems.

The system to be analysed must be given as a block diagram formed by mini fault trees of individual system components. The programs were written in LISP programming language and run on a PDP8 computer with 8k words of storage.

This report describes the methods used and gives a detailed description of the program construction and working.

Copies to

*

Permanent address:
Institute for Electrical Power Research (VEIKI)
H-1051 Budapest
Zrinyi-street 1
HUNGARY

Available on request from the Library of the Danish Atomic Energy Commission (Atomenergikommissionens Bibliotek), Risø, DK-4000 Roskilde, Denmark
Telephone: (03) 35 51 01, ext. 334, telex: 43116

Fi 25-204

ISBN 87-550-0442-3

CONTENTS PAGE

Introduction

Fault tree and consequence diagram analyses have recently received widespread interest as methods for reliability and safety analysis of complex systems. Haasl's paper (1) can be considered as the starting point of fault tree technique applications and Nielsen's report (2) indicates the beginning of cause-consequence charts' use in practice. In the field of fault trees, however, after an optimistic start, there has been some scepticism. The main problems were the cost and time aspects of constructing complex fault trees; to consider all failure combinations; and to obtain proper failure data; and to find qualified staff with experience in fault tree method, probability analysis, and system operation. By the late 60's and early 70's several of these problems have been overcome, but the fault tree and consequence diagram construction is still the most critical point of the analysis procedure. The state of the art of fault trees and CCD's are summarized in Fussell/Powers/Bennets' joint paper (3) and Nielsen's report (4), respectively.

In order to reduce the cost of adequate diagram construction and to avoid oversights of some failure sources or consequences, automated treatment is required. On the other hand, it has some disadvantages, e.g. human errors and environmental effects cannot be considered but it can be a rapidly executed initial procedure, to be followed by a more detailed fault tree or consequence analysis. Up to the present algorithms published on automated fault tree or consequence diagram construction are rather limited.

Fussell's method (5) uses mini fault trees of different components, the system fault tree is created by their consistent connections. His technique has been implemented on computers for electrical systems. Powers and Tompkins (6) use input-output component models for fault tree constructions, where the component's normal or failure state can be identified by the actual input-output process variable values. Lapp and Powers (7) employ digraph models for components which describe the normal, failed and conditional relations among variables and events. Their computer program was applied to chemical processes.

Methods for obtaining consequence diagrams are given by Taylor (8, 9). His method uses algebraic equations for components to describe their normal or failed operation. The application of his algorithm on computers is in progress.

Generally both the automatic fault tree development and the automatic consequence diagram constructions require three main steps:

- to find a proper system or component modelling method which is suitable for computer programming,

- to develop an algorithm for fault tree and consequence construction,

- to implement these algorithms on computers.

In this paper algorithms and programs for automatic fault tree and consequence diagram construction are presented. The programs were written in a LISP dialect and developed for a PDP8 computer with 8k. For plant component models input/output and state transfer functions formalized as mini fault trees are used, the algorithms work with their causal links which form the system model.

1. Unit model

1.1 Failure transfer functions

Both consequence and fault tree programs use individual plant component failure transfer functions. The unit models receive input events/conditions as well as state information and depending on combinations of these, the output events can be determined. The transfer functions are considered as component mini fault trees describing the possible failure modes of the unit. To determine the mini fault trees thorough component failure mode and effect analysis (FMEA) is required. The results of this analysis, i.e. failure transfer functions are formalised as Boolean expressions using OR and AND gates to describe the connection between input and output events.

The structure of mini fault trees for the programs presented here is as follows:

```
<Transfer function>::=(TF<TF>)
<TF>::=((STF1 <STF1>)(STF2 <STF2>)...)
<STFi>::=((OR(AND<Input Event/Cond>)(AND<Input Event/Cond>)...)
          (AFTER Ø <Mark> <Immediate event>)
          (AFTER <Time delay> <Mark> <Delayed event>))
<Mark>::=SIGNEV|LASTEV|PUNEV
```

The failure transfer function consists of a set of sub-transfer functions (STFi). Each sub-transfer function has an input and output part, the input part contains the OR/AND combinations of input events/conditions and state variables, the output part involves an immediate and delayed event. The immediate output event has zero time delay, the delayed output event has to have a non-zero time delay. Either of them may be missing from the given sub-transfer function. In both of them a marker is used for indicating a significant (SIGNEV), a last-in-chain (LASTEV), or minor puny (PUNEV) event for display selection purposes.

The structure of input combinations and output events is as follows:

⟨Input Event/Cond⟩::=⟨Input 1⟩⟨Input 2⟩....⟨Input n⟩
⟨Input n⟩::=⟨SEn⟩|⟨NSn⟩|⟨ECn⟩
⟨Output Event⟩::=⟨EC1⟩⟨EC2⟩....⟨ECm⟩
⟨SEi⟩::=(SE ⟨VNi⟩→⟨VVi⟩)
⟨NSj ::=(NS ⟨VNj⟩ = ⟨VVj⟩)
⟨ECk⟩::=(EC ⟨VNk⟩ ⟨Relation⟩ ⟨VVk⟩)
⟨Relation⟩::=→|=

In this structural description a distinction is made between spontaneous events (SE), normal state information (NS) and events/ conditions (EC) appearing between components. To make clearer the difference of events and conditions "→" and "=" relations marks between variable names (VN) and their values (VV) are used.

1.2 Unit descriptions

Although the failure transfer functions determined by FMEA analysis form the critical part of system unit descriptions, some further information is needed to describe the physical connections between individual components, and to make effective programming possible.

The complete unit description which is applicable for both algorithms contains the following information.

⟨Unit description⟩::=(⟨Component name⟩
⟨Failure transfer function⟩
⟨Preceeding connected component list⟩
⟨Following connected component list⟩
⟨Spontaneous event list⟩
⟨Normal state⟩
⟨Variable list⟩)

⟨Preceeding connected component list⟩::=
(PC ⟨PCName 1⟩ ⟨PCName 2⟩...)
⟨Following connected component list⟩::=
(FC ⟨FCName 1⟩ ⟨FCName 2⟩...)
⟨Spontaneous event list⟩::=(SE ⟨SE1⟩⟨SE2⟩...⟨SEi⟩)
⟨Normal state⟩::=(NS ⟨NS1⟩⟨NS2⟩...⟨NSj⟩)
⟨Variable list⟩::=(VR ⟨EC1⟩⟨EC2⟩...⟨ECk⟩)

The preceeding/following connected component lists can be obtained from the system block diagram; the spontaneous event list, the normal state, and the variable list can be filtered from the failure transfer function. This unit description method has some advantages and some drawbacks.

Drawbacks:
- the unit descriptions are dependent of the system being analyzed (as PC/FC lists are contained),
- redundant information is involved (in TF and SE/NS/VR).

Advantages:
- clear, complete, and easy-to-change structure,
- efficient computer programs can be developed,
- a library data of transfer functions can be set up which is independent of the system structure on a large scale. In current state this independency is restricted by the condition that the names of input-output variables in a causal connection must be identical.

This means that in order to get well developed programs and shorter running times, a certain amount of surplus storage capacity is necessary. The detailed specific rules to create unit descriptions are summarized in Table 1.

Table 1 Rules specified for unit model descriptions

1. Only atomic, non-zero component name is allowed.
2. Only atomic, non-zero variable name is allowed.
3. Only atomic variable value is allowed.
4. Each unit indicated in PC/FC list must be figured on CL.
5. A variable might appear at the same sub-TF's input and output only if it is an internal state variable.
6. If a component has an internal feedback variable, the variable name must be in its PC/PC list.
7. Several identical sub-TF's output events may occur in a TF, but their time delays must be different.

x 8. Each variable of TF inputs must appear on VR list of component in question, except normal state.

x 9. Each variable of TF outputs must appear on FC's VR lists.

x10. The sequence of variables in VR list should be adequate to the sequence of components in FC list.

xx11. The sequence of variable types in a sub-TF's input combinations must be: first SE/NS and EC.

xx12. If in a sub-TF's input combination an SE/NS-type variable occurs, the component name must be on its PC list and on the first place.

Note:

Rules signed by x are raised by consequence diagram program, signed by xx are raised by fault tree program.

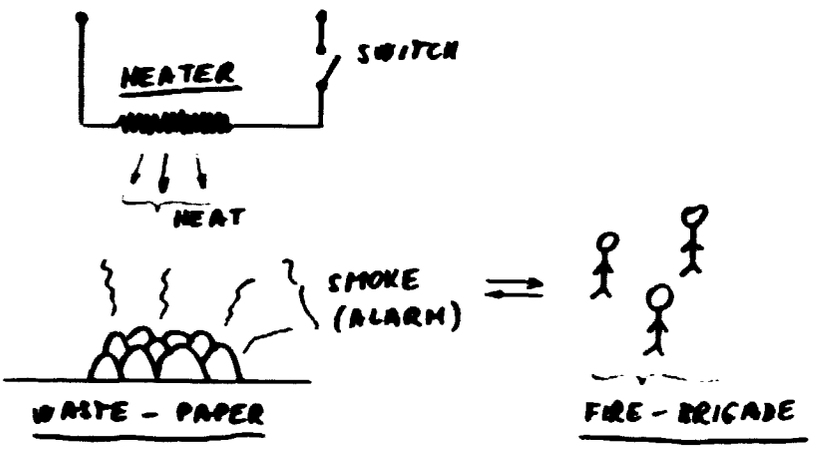
It must be noted that unit descriptions used for either consequence diagram or fault tree construction alone, can be significantly simplified. Namely, for consequence program the preceeding connected component list (PC), spontaneous event list (SE) and normal state (NS) can be eliminated, for fault tree program the following connected component list (PC) and variable list (VR) can be omitted.

1.3 Illustrative example

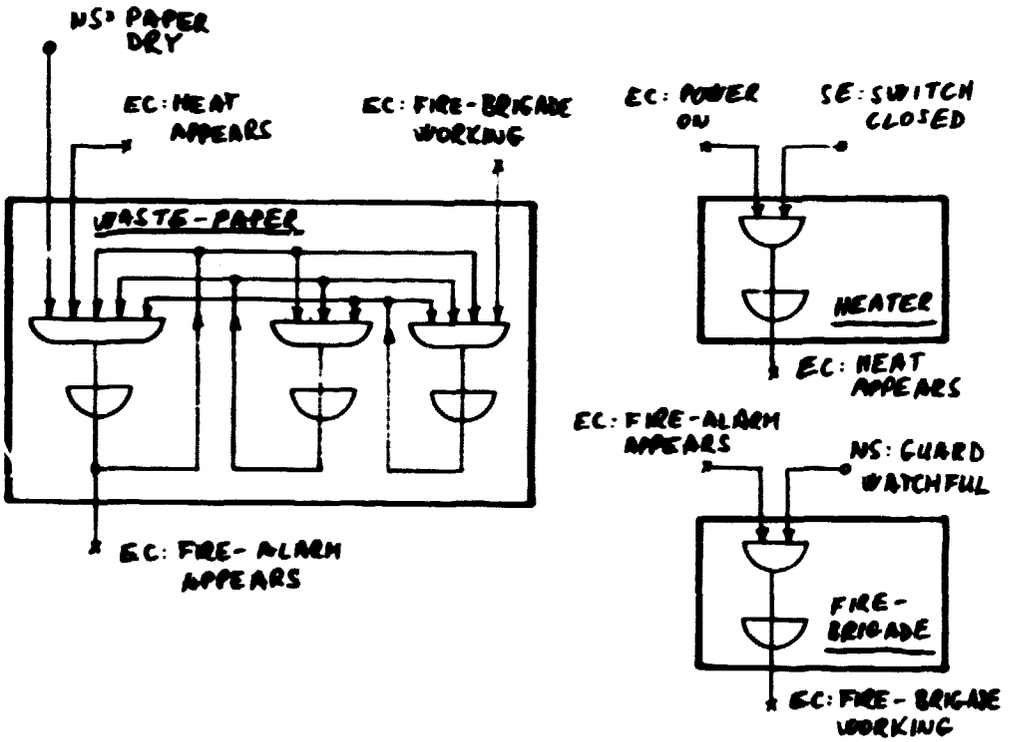
To illustrate the method, a simple example with general event transfer functions is given in Fig. 1. The system consists of three components: a HEATER, a pile of WASTE-PAPER, and a FIRE-BRIGADE.

The HEATER can be switched on by SWITCH. After the heater has been turned on, there will be a time delay Δt_p, after which a FIRE-ALARM may occur, but only if the WASTE-PAPER is DRY. The FIRE-BRIGADE which is probably WATCHFUL is alarmed by the papers SMOKING and tries to extinguish the fire. (WASTE-PAPER state = BURNING, SMOKING is changed to EXTINGUISHED). If the fire-brigade is not quick enough ($\Delta t_g > t_p$), the papers may be COMPLETELY-BURNT.

The component descriptions are summarized in Table 2. It can be seen that in order to get a unified library data of component descriptions for both programs, a relatively complex data structure was chosen, but simultaneously it yields a profit of clear and efficient program outline.



a.) Block scheme



b, Mini fault trees

Fig. 1. Unit model example (HEATER/WASTE-PAPER/FIRE-BRIGADE).

Table 2. Example of unit description

```
((HEATER

(TF(STF(OR(AND(SE SWITCH → CLOSED)(EC POWER = ON)))
       (AFTER 7 SIGNEV (EC HEAT → APPEARS))))
(PC HEATER)
(FC WASTE-PAPER)
(SE (SE SWITCH → CLOSED))
(NS)
(VR (EC POWER = ON)(EC SWITCH → CLOSED)))

(WASTE-PAPER

(TF(STF1(OR(AND(NS PAPER = DRY)(EC HEAT → APPEARS)))
        (AFTER Ø SIGNEV (EC PAPER → SMOKING)(EC FIRE-ALARM → APPEARS))
        (AFTER 5 SIGNEV (EC PAPER → BURNING)))
   (STF2(OR(AND(EC PAPER → BURNING)))
        (AFTER 20 LASTEV (EC PAPER → COMPLETELY-BURNT)))
   (STF3(OR(AND(EC PAPER → SMOKING)(EC FIRE-BRIGADE → WORKING))
            (AND(EC PAPER → BURNING)(EC FIRE-BRIGADE → WORKING)))
        (AFTER 3 SIGNEV (EC PAPER → EXTINGUISHED))))
(PC WASTE-PAPER HEATER)
(FC WASTE-PAPER FIRE-BRIGADE)
(SE)
(NS (NS PAPER = DRY))
(VR (EC HEAT → APPEARS)(EC FIRE-BRIGADE → WORKING)))

(FIRE-BRIGADE

(TF(STF(OR(AND(NS GUARD = WATCHFUL)(EC FIRE-ALARM → APPEARS)))
       (AFTER 10/30 SIGNEV (EC FIRE-BRIGADE → WORKING)))
(PC WASTE-PAPER)
(FC WASTE-PAPER)
(SE)
(NS (NS GUARD = WATCHFUL))
(VR (EC ALARM → APPEARS))))
```

2. Consequence diagram construction

The consequence diagram is an event-sequence diagram, which relates the input events of a system into its output events. During the consequence analysis procedure a tracing work is done, where at each step taking the actual system state into consideration the current component's input event is implied on the component and its related output event is deduced. In the next step this output event is considered as input event for the selected next components, thus determining the possible event chains until last chain events or unfulfilled input conditions are reached.

2.1 Consequence diagram construction algorithm

The consequence analysis procedure can be considered as a transformation of a system block diagram into an event sequence diagram. The descriptions of individual system units must be given in the form of failure transfer functions and some other additional information described in Chapter 1.2. The system may contain components connected simply in series or components with internal/external feedback and feedforward loops. A single series block diagram is converted into a simple series event chain or if the components have memory (i.e. its output event contains a delayed event with non-zero time delay) the chains may branch. The components may have several connected components which are affected by the current output event of the preceeding component. Each of the activated following components initiates a new branch or branches (components with memory). Similarly, the physical series branches of a block diagram can be transferred into a simple series event chain or several series event chains depending on the causal interconnections. Feedback/feedforward loops containing components with several input/output connections and time delays draw special attention. Several input connections indicate several input conditions which must be investigated to determine an event propagation through the component. Feedback loops with time delays produce several delayed event chains of

one input event, which makes the consistency checking very important. Details and some other aspects of block/consequence diagram transformation (e.g. multiple failures) are described in (8).

To perform event tracing tasks in complex systems containing either simple series or complicated loop block diagrams a simple algorithm was developed. The main steps of the algorithm are presented in Table 3, the detailed description in Table 5, in Appendix 1.

2.2 Program for consequence diagram construction (CONSEQ)

To adapt the shown algorithm on computers a dialect of LISP language was chosen which made the list processing of data possible. Structured programming techniques were applied to give a possibility of easy modifications and to yield well-arranged programs.

2.2.1 General program structure and description

The general block structure of the developed program is given in Fig. 2.

The system information is stored as component description list CL. The program starts with creating a data-field containing all the component information in a clear, easy-to-handle way, i.e. the data-field is set up by an object-set. Each object is related to a component and its attributes comprehend the unit description in a slightly modified, internal data structure. To manipulate the data-base (to select an object or an attribute, to update attribute values, etc.) a program system for heuristic programming (10) was used. The object structure is as follows:

Table 3 Consequence mapping algorithm

1. Get initial input events marked with their time and influenced component's name.

2. Select earliest output event and active component. If there are no more output events, stop.

3. Find the affected following components.

4. Select randomly a following component. If there are no more, go to 2.

5. Check for match between the selected components' input events and current input conditions/internal state. If there is no match, go to 4.

6. Deduce its current intermediate and/or delayed output events and their real time of occurrence.

7. Update internal state condition.

8. Go to 4.

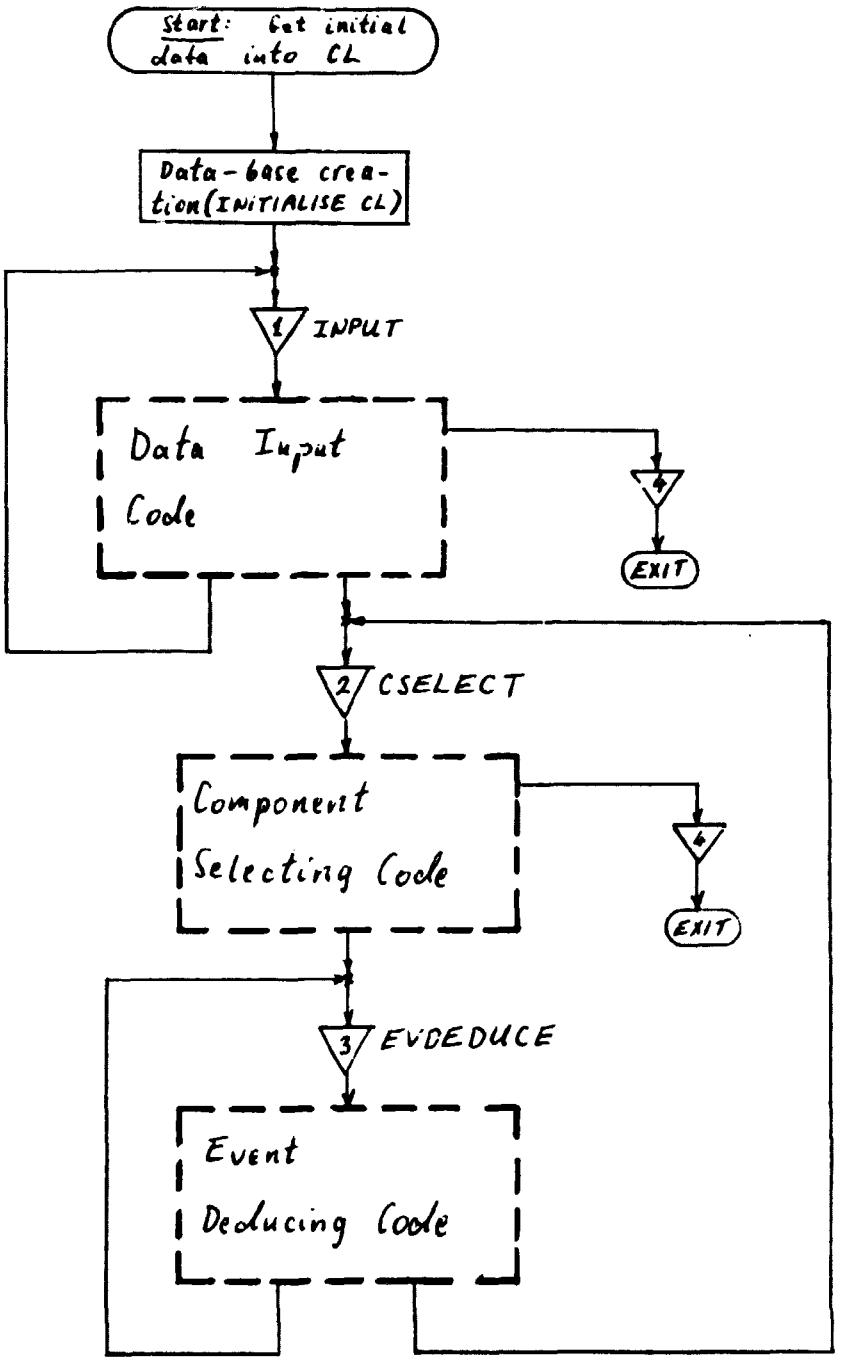


Fig. 2 Consequence program block scheme.

```
⟨Object structure⟩::=⟨Internal component description⟩
⟨Internal component description⟩::=
     ((NAME ⟨Component name⟩)
      (TF SS ⟨D⟩ (STF1....)(STF2....)....)
      (PC SS ⟨D⟩ ⟨PCN1⟩ ⟨PCN2⟩  ....     )
      (FC SS ⟨D⟩ ⟨FCN1⟩ ⟨FCN2⟩  ....     )
      (SE SS ⟨D⟩ ⟨SE1⟩⟨SE2⟩     ....     )
      (NS SS ⟨D⟩ ⟨NS1⟩⟨NS2⟩     ....     )
      (VR SS ⟨D⟩ ⟨EC1⟩⟨EC2⟩     ....     )
      (FE SS ⟨D⟩ (Ø ⟨Mark⟩ ⟨Immediate event⟩)
                 (⟨Time delay⟩ ⟨Mark⟩ ⟨Delayed event⟩)))
```

where SS is an indicator of a Simple Set, FE contains the component's actual following output events and ⟨D⟩ is a Dummy value resulting from the used heuristic programming technique.

The main program can be divided into three main parts: data input, component selecting and event deducing codes. The Data Input Code receives the initial input events, their times and the influenced component's names to start the event tracing procedure. Each initial component name is checked to be a system component or not and in case of correctness is placed on the active component list (ACL) marked with the initial event time. ACL indicates the active components in absolute time order, thus enabling the consequence evaluation of multiple failures. Its structure corresponds to an object structure having one attribute:

```
⟨ACL⟩::=((NAME AL)
         (AL SS ⟨D⟩ (⟨Time 1⟩ ⟨ACN1⟩ ⟨ACN2⟩....)
                    (⟨Time n⟩ ⟨ACNi⟩ ⟨ACNi+1⟩....)))
```

The Component Selecting Code searches for components to be currently investigated. First the "main" component with earliest output event time is selected from ACL, its output event has been stored in its attribute FE. In the next step the affected (current) following components are selected from its connected following component list which are indicated in attribute FC.

The basis of this selection is a match-tracing procedure which looks for an identity between the main component's output event and the connected components' possible input events/conditions.

The Event Deducing Code is the most central part of the consequence program which determines all the possible local output events and their time values. For simplification the elements of the current input events/conditions and current component state are placed in a global variable list VR, and the deducing procedure is reduced to a systematical match-search mechanism between VR and the input combinations of the current following component' transfer functions. After the immediate/ delayed output events and their time having been evaluated, the components' future event list FE and the active component list ACL are updated to prepare them for the next calculation cycles.

The connections between the above described codes are shown on Fig. 2, their detailed structure is presented on Fig. 9-11, in Appendix 1. The program outline is set up to meet the requirements of a general LISP-8 program. This framework is illustrated in Appendix 3, and the structure of the internal global variables in Table 6, in Appendix 1.

2.2.2 Subroutine description

Both the consequence and fault tree codes are written by structured programming technique, i.e. all the separatable tasks are comprised within subroutines and only their interfaces are involved in the main routine. The functional description of the consequence subroutines, interpretation of their arguments and outputs are given below.

INITIALISE X

This routine creates the object on the active component list ACL and on the data-base using the input data description list CL.

Arg. X must be set to CL.
It returns with NIL (DATA-ERROR) in case of empty CL or with CL otherwise.

CHECK X Y

The routine checks the existence of a component marked with its name in component list CL.
Arg. X must be set to CL, Y to component name.
It returns with NIL (DATA-ERROR) if the component has not been found in CL (or empty CL) or with the component name otherwise.

CSEL X

This routine selects the active component name with earliest output event time from the active component list ACL.
Arg. X must be set to ACL.
It returns with NIL if ACL is empty or with (<Time> <ACName>) pair.

CDEL X Y

This routine deletes a given component name from ACL in function of the specification:
- if the entry specification is a (<Time> <CName>) pair, the component name is only deleted from the specified time-branch,
- if the entry specification is a <CName>, it is deleted from all existing time-branches.

Arg. X must be set to ACL, Y to the entry specification.
It returns with the modified ACL.

CINS A X Y

The routine appends a component name to the end of a specified time-branch of ACL.
Arg. A must be set to ACL, X to the event time, and Y to the component name.
It returns with the modified ACL.

FFEV X

The routine selects the earlier output event from a component's future output event list FE and deletes it.
Arg. X must be set to component description.
It returns with the selected output event or with NIL if FE is empty.

FCSEL A X Y

This routine determines the current following components of a main component which are affected by its output event.
Arg. A must be set to CL, X to the main component description, and Y to the output event.
It returns with the affected following component list.

INUPDT X Y

The routine updates the values of variables contained in the variable list VR of a component. Arg. X must be set to component description, Y to the actual variable list.
It returns with the new set of variables VR.

DEDUCE X

This routine deduces the possible output event of an active component, i.e. of a component whose variable values are updated by the actual input events/conditions.
Arg. X must be set to the activated component description.
It returns with the founded output event or NIL if there is no fulfilled transfer function input combination.

2.2.3 Input/output

Input

The system block diagram is stored as a unit description list. The multiple input failure event descriptions, i.e. initial component names, their input failure events marked with times must be reported through display keyboard. The initial data transfer is over by giving Ø component name.

```
┌─→COMPONENT: (<Comp. name>)
│  TIME:      (<Event time>)
│  EVENT:      <Input event>
└────*
```

Output

In current version the information on event occurrence chains are presented on teletype and display screen. On the teletype each investigated component's name, marked with event time, output event, active following components' names associated with their selected output events are printed out. This information serves for manual or automatic consequence diagram drawing.

```
MAIN-COMPONENT: (<Time> <Comp.name>)
OUTPUT-EVENT:    <Output event>
FOLLOWING-COMPONENTS:
   (<FCN1> <FCN2> ...<FCNi>)
    <FCN1> <FC1 Output event>
    <FCNi> <FCi Output event>
```

On the screen only the significant output events (SIGNEV) or last chain events (LASTEV) and the relating main component name with time are displayed.

```
MAIN-COMPONENT: (<Time> <Comp. name>)
OUTPUT-EVENT:    <Output event>
```

2.3 An example

To illustrate the results gained by the consequence diagram construction program CONSEQ, the outputs of HEATER/WASTE-PAPER/FIRE-BRIGADE example shown in Chapter 1.3 are presented on Fig. 3-4.

Fig. 3 contains the output got by COMPLETELY-BURNT WASTE-PAPER, i.e. the FIRE-BRIGADE's interaction was too late to extinguish the fire ($\Delta t_B = 30 > \Delta t_p = 20$). In Fig. 4 the elements of consequence diagram gained by NON-BURNT WASTE-PAPER are shown. In this case the FIRE-BRIGADE was quick enough to extinguish the paper-fire ($\Delta t_B = 10 < \Delta t_p = 20$).

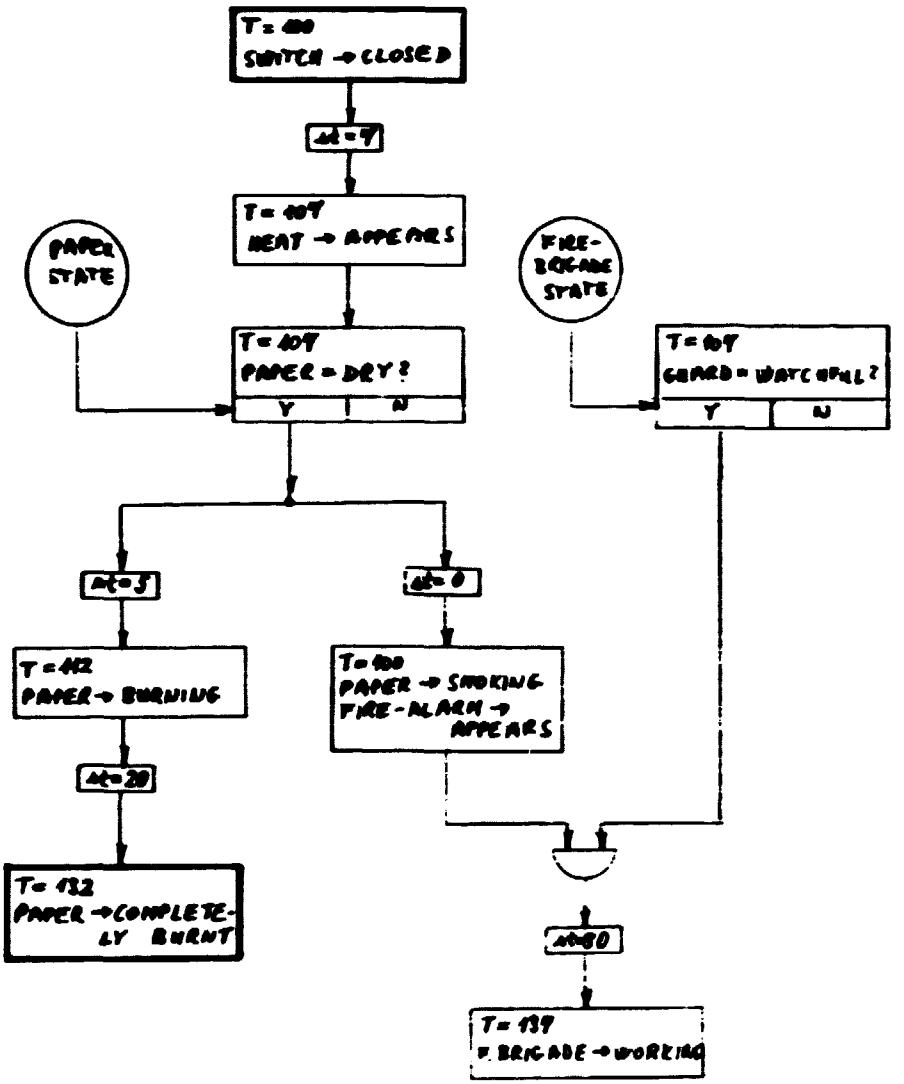


Fig. 3. Consequence diagram of BURNT WASTE-PAPER.

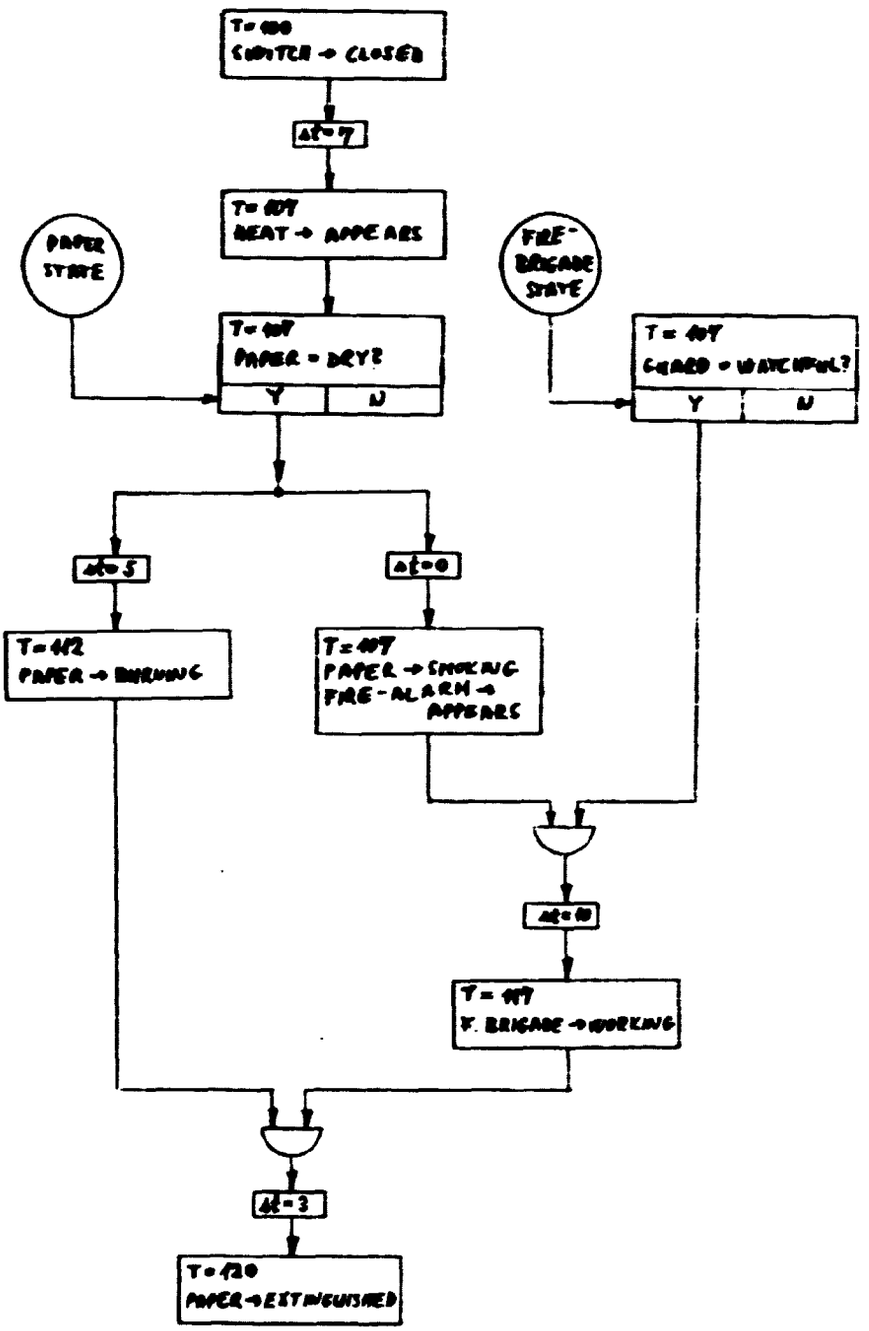


Fig. 4. Consequence diagram of EXTINGUISHED WASTE-PAPER.

3. Fault tree construction

The fault tree is a clear, graphic representation of a logical function which relates a specified undesired event to its contributing events. The output event is often called as TOP event or system failure event, and its causes as primary or spontaneous events. In one fault tree only one TOP event is emphasized and several primary events are presumed.

3.1 Fault tree construction algorithm

The fault tree construction starts with the definition of an undesired event and a backward tracing is carried out to map the combinations of possible input conditions/events and component state which can cause the output failure event. The tree branches are terminated if spontaneous input events or normal unit states are reached.

The necessary unit and system information for fault tree construction is described in Chapter 1, but now the system may contain only internal loops, external loops cannot be handled. Special attention is directed towards event timing, i.e. sequential fault trees are handled. The automatic procedure cannot at present treat environmental and human aspects of failure.

The main steps of the developed algorithm are described in Table 4, its details in Table 7, in Appendix 2. The algorithm is based on Fussell's method (5), the deviations mainly concern event timing and description of components with memory.

3.2 Program for fault tree construction (CAUSE)

The program for fault tree algorithm was written in LISP by structured programming technique.

Table 4 Backward tracing algorithm

1. Get initial output event marked with time and component name, go to 4.

2. Select an input variable combination indicated in old sub-branch. If there are no more, go to 7.

3. Find current preceeding components and their output events. If there are no output events, go to 2.

4. Select an output event. If there are no more, create a new AND-branch, go to 2.

5. Search for possible new input variable combinations which can lead to the selected output event and calculate their time of occurrence.

6. Create a new OR-branch, go to 4.

7. Get currently created set of AND-branches. If it is empty, prune event tree, go to 9.

8. Create new tree-branch. If only primary events are involved, build event tree.

9. Get next sub-branch of latest tree-branch. If there is no more, build final fault tree, exit.

10. Go to 2.

3.2.1 General program structure and description

The general block structure of the developed program is given on Fig. 5. The program starts with basic data-field creation by using system information stored as unit description list in CL. The produced component related objects have the same structure as presented in Chapter 2.2.1, but no following output event lists are involved.

The main program can be divided into three major parts: cause-event searching, new branch making and event tree making codes. The Cause-Event Searching Code gets the initial output event marked with time and component name to start the backward tracing. First the possible subtransfer functions and their input variable combinations are selected for the given output event. For proper selection a preliminary consistency checking is carried out to delete mutually exclusive simultaneous events and which can be completed by using a consequence checking procedure. The input combinations may generally be built of spontaneous events and normal states of the investigated component or input events/conditions coming from the previous connected components. To find these categories of input variables a component related type selection is carried out and the result is placed on a stack (IV). The separated spontaneous events/normal state are built in the final part of fault tree (selected final input - SI), the input events/conditions are regarded as possible output events of previous components (selected temporary output - SO) and are inserted in the temporary part of fault tree for further investigations.

The relation between an output event and its input variable combinations, i.e. the current part of the fault tree to be constructed is created by using AND- and OR-gates involved in unit mini fault trees. The New Branch Making Code receives the formerly established logical functions of input/output variables and produces a new preliminary branch of the fault tree. Each prelinimary tree-branch is stored on a Branch Stack BS and is built up of several sub-branches. To make their identifications

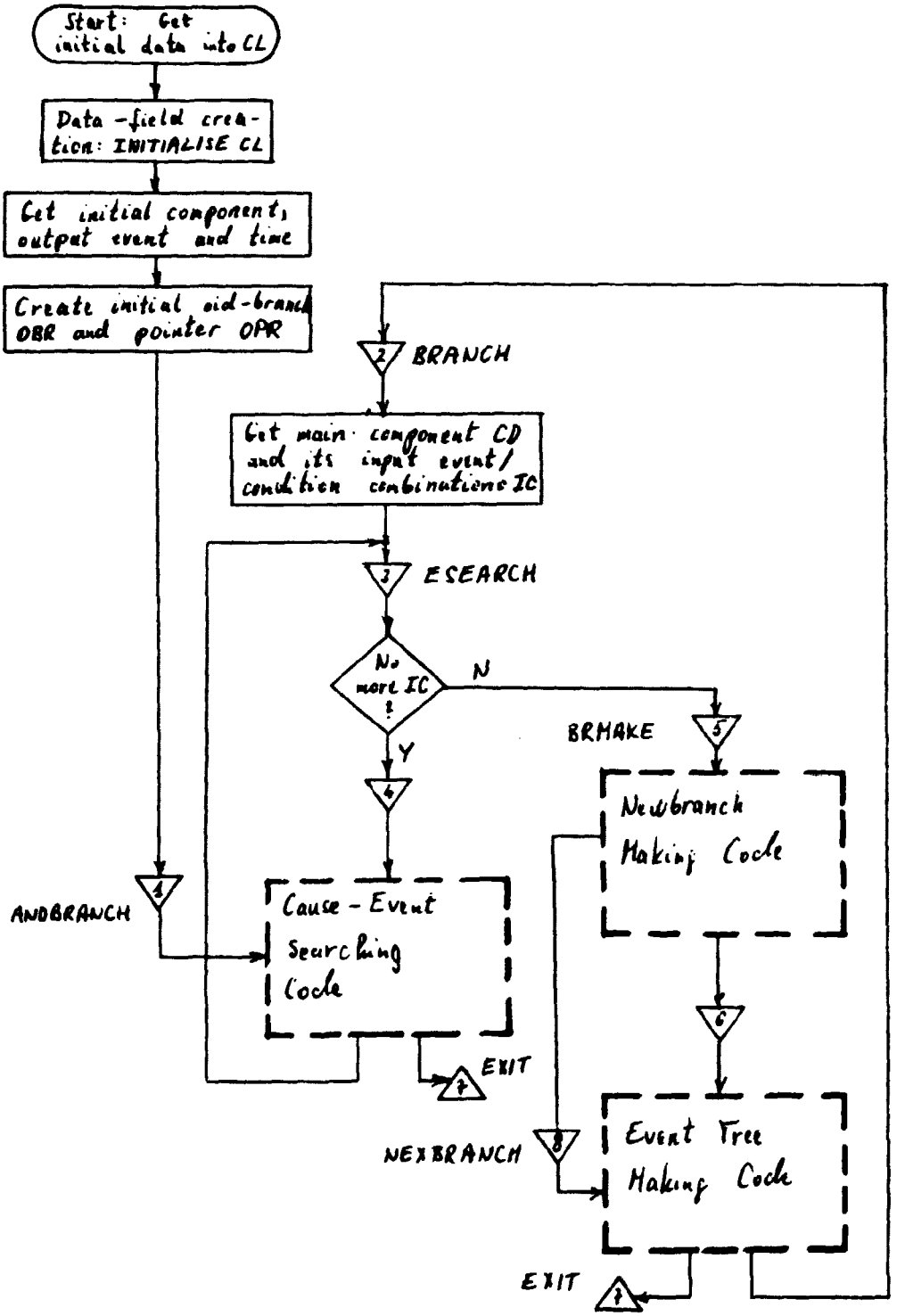


Fig. 5 Fault tree program block scheme.

easier, pointers and sub-pointers are introduced and placed on a Pointer Stack PS. The structure og tree-branches and pointers are shown on Fig. 6-7.

The main manipulation and supervision of tree-branches is carried out by the Event Tree Making Code. A preliminary tree-branch consists of two main parts: a tree-branch pointer and a set of sub-branches. The sub-branches may be complete or preliminary sub-branches. If all the sub-branches in a tree-branch are complete, i.e. they are traced back until spontaneous events/normal states are reached, the tree-branch is inserted into its "mother-branch", i.e. into a tree-branch for which itself is a sub-branch. In other words, a tree-branch is complete and not treated further if all its sub-branches are complete or is preliminary if one of its sub-branches is preliminary.

The backward search is continued until only one complete tree-branch is found, this represents the final fault tree. The detailed block scheme of the above described codes are given on Fig. 12-14, the structure of mini internal global variables in Table 8, in Appendix 2.

3.2.2 Subroutine description

In the following the functional description of fault tree subroutines, the interpretation of their arguments and outputs are given.

INITIALISE X

The same as of consequence program described in Chapter 2.2.2.

CHECK X Y

The same as of consequence program described in Chapter 2.2.2.

INSEL X Y

This routine carries out a component related type-selection

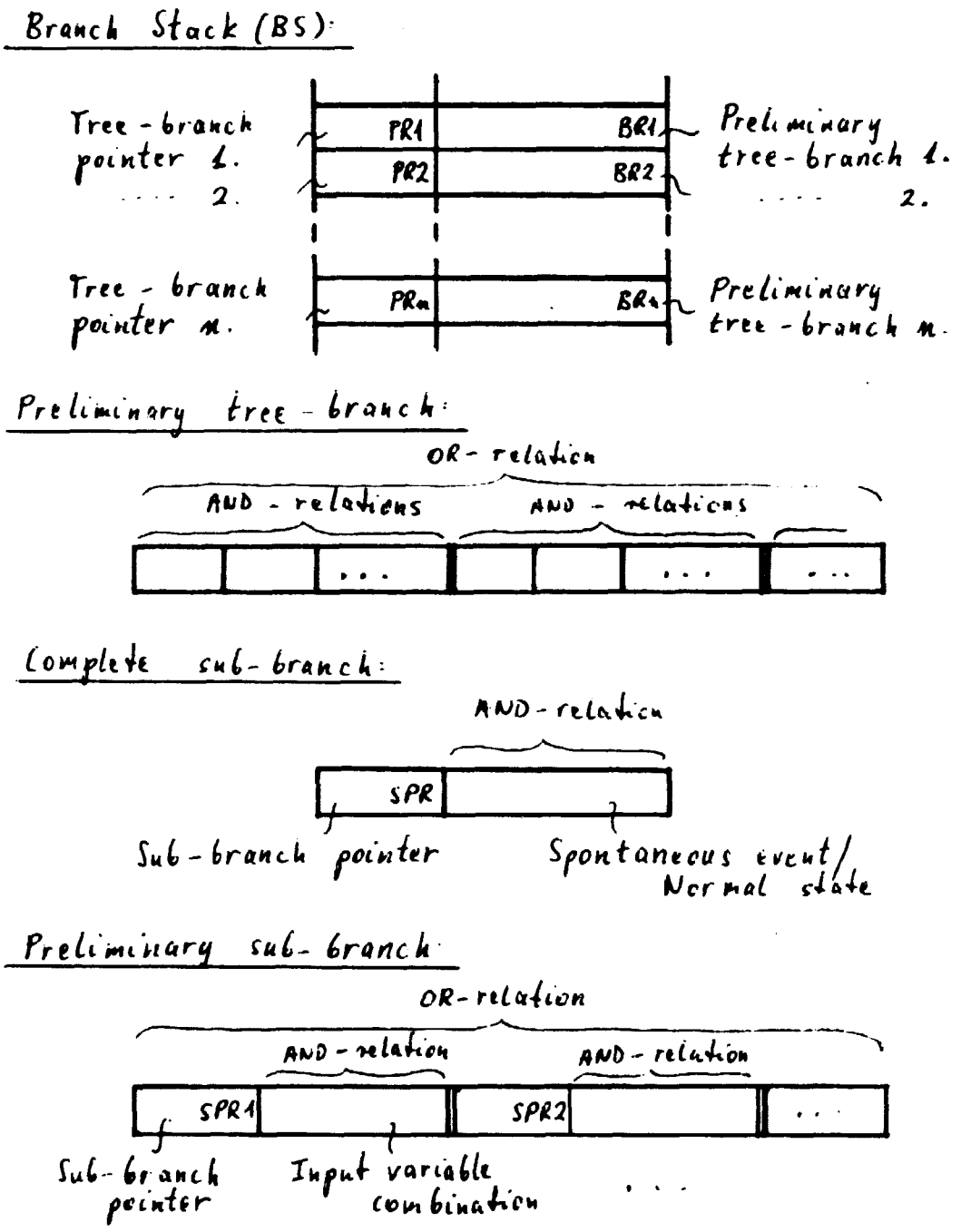


Fig. 6 Structure of tree-branches.

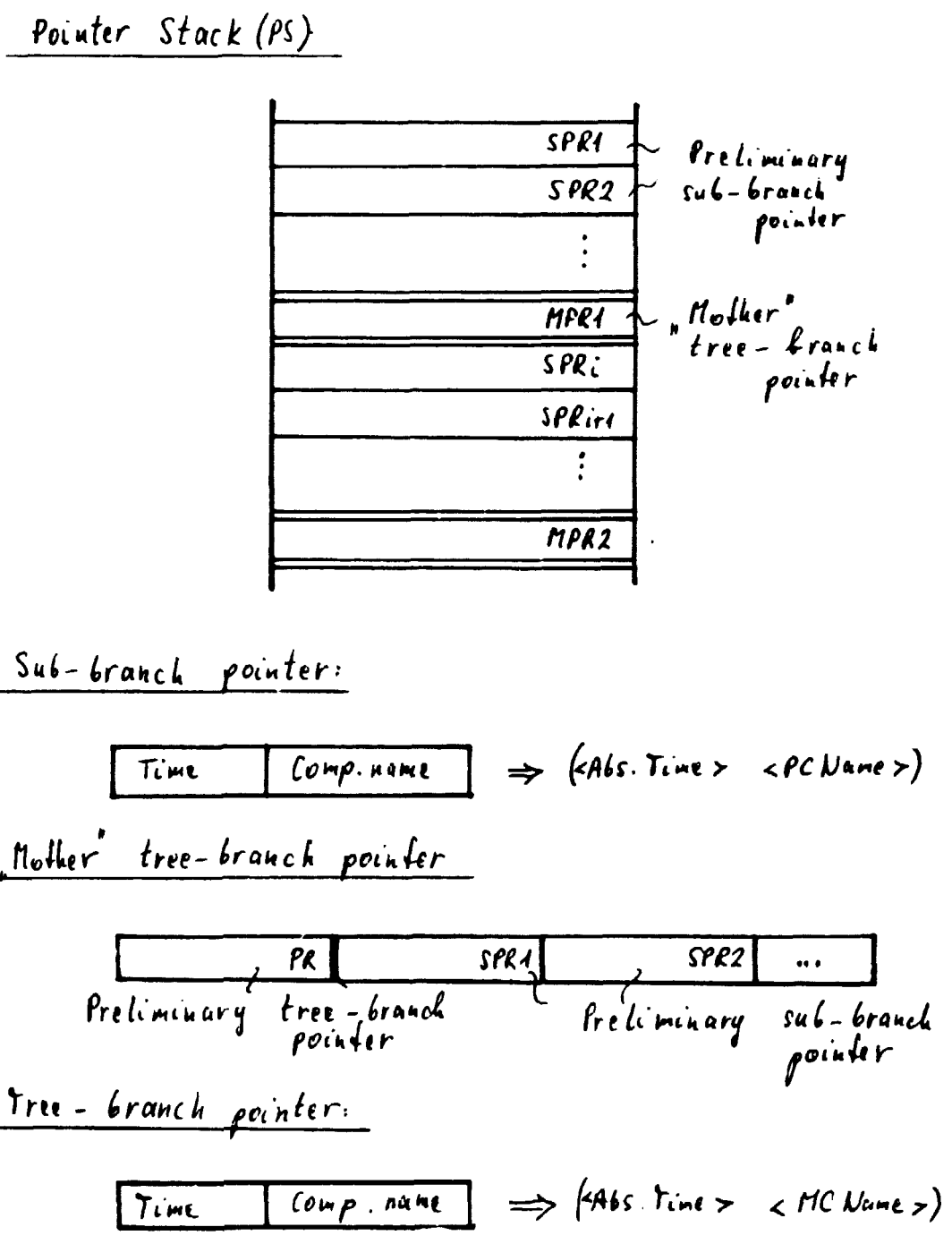


Fig. 7 Structure of pointers.

of input variable combinations into spontaneous events/ normal states and input events/conditions.
Arg. X must be set to main component description, Y to its input variable combination.
It returns with NIL (DATA-ERROR) if an input variable has not been found in the main component's SE/NS lists or among the previous components' output variables, otherwise it returns with the component related distributed variable list.

EvSEARCH X Y

This routine searches for possible input variable combinations of a component related to a supposed output event.
Arg. X must be set to component description, Y to output event.
It returns with NIL if the output event turned out to be an erroneous event or with input combinations marked with time fast, otherwise.

TMAKE T X Y

The routine calculates the absolute time value of input variable combinations.
Arg. T must be set to initial time value, X to component name, and Y to input combinations marked with their time fast.
It returns with input variable combinations marked with component name and absolute time.

SELBRANCH X Y

This routine selects a sub-branch marked with pointer from its "mother" tree-branch.
Arg. X must be set to sub-branch pointer, Y to tree-branch.
It returns with NIL if the pointed sub-branch has not been found in the given tree-branch or with the selected sub-branch otherwise.

INSBRANCH X Y

This routine inserts a sub-branch into the pointed part of its "mother" tree-branch (into head of BS). If the sub-branch

to be inserted is empty, the mother tree-branch is pruned.
Arg. X must be set to sub-branch, Y to its pointer.
It returns with NIL if the pointed sub-branch does not belong to the head of BS or with modified BS otherwise.

PRUNETREE X

This routine reduces the event tree until it is possible, i.e. eliminates empty sub-branches and empty tree-branches from BS.
Arg. X must be set to the pointer of the empty sub-branch to be eliminated at first.
It returns with NIL if pruned event tree is empty or with BS otherwise.

NEWPTR X

This routine creates a "mother" pointer of a tree-branch.
Arg. X must be set to tree-branch.
It returns with new "mother" tree-branch pointer.

INSPTR X

This routine inserts a "mother" pointer as well as its sub-branch pointers into the pointer stack PS.
Arg. X must be set to "mother" pointer.
It returns with extended PS.

DELPTR X

The routine deletes given sub-branch pointers from their "mother" tree-branch pointer.
Arg. X must be set to sub-branch pointer set to be deleted (if X=Ø, the "mother" pointer is entirely deleted).
It returns with NIL if an indicated sub-branch pointer has not been found in its "mother" pointer or with reduced PS otherwise.

3.2.3 Input/output

Input

The system information is stored as a unit description list. The system failure / undesired event description, i.e. initial component name, its output failure event and time must be communicated through display keyboard:

COMPONENT: (⟨Comp. name⟩)
TIME: (⟨Time⟩)
EVENT: ⟨TOP event⟩

Output

The input system failure / undesired event description as well as the resulted Boolean fault tree expression are displayed on screen:

COMPONENT: ⟨Comp. name⟩
TIME: ⟨Time⟩
EVENT: ⟨TOP event⟩
FAULT TREE: ⟨Fault tree⟩

The final fault tree consists of spontaneous events/normal states marked with components names and time values and connected through AND/OR gates:

⟨Fault tree⟩::=((AND(OR(AND⟨Tree-branch 1⟩)(AND⟨Tree-branch⟩ 2)...)))
⟨Tree-branch i⟩::⟨Sub-branch 1⟩⟨Sub-branch 2⟩...
⟨Sub-branch j⟩::=(⟨Spontaneous event/Normal state⟩) |
(OR(AND⟨Tree-branch k⟩)(AND⟨Tree-branch k+1⟩)...)

3.3 An example

For illustration the example of HEATER/WASTE-PAPER/FIRE-BRIGADE system described in Chapter 1.3 is presented on Fig. 8.

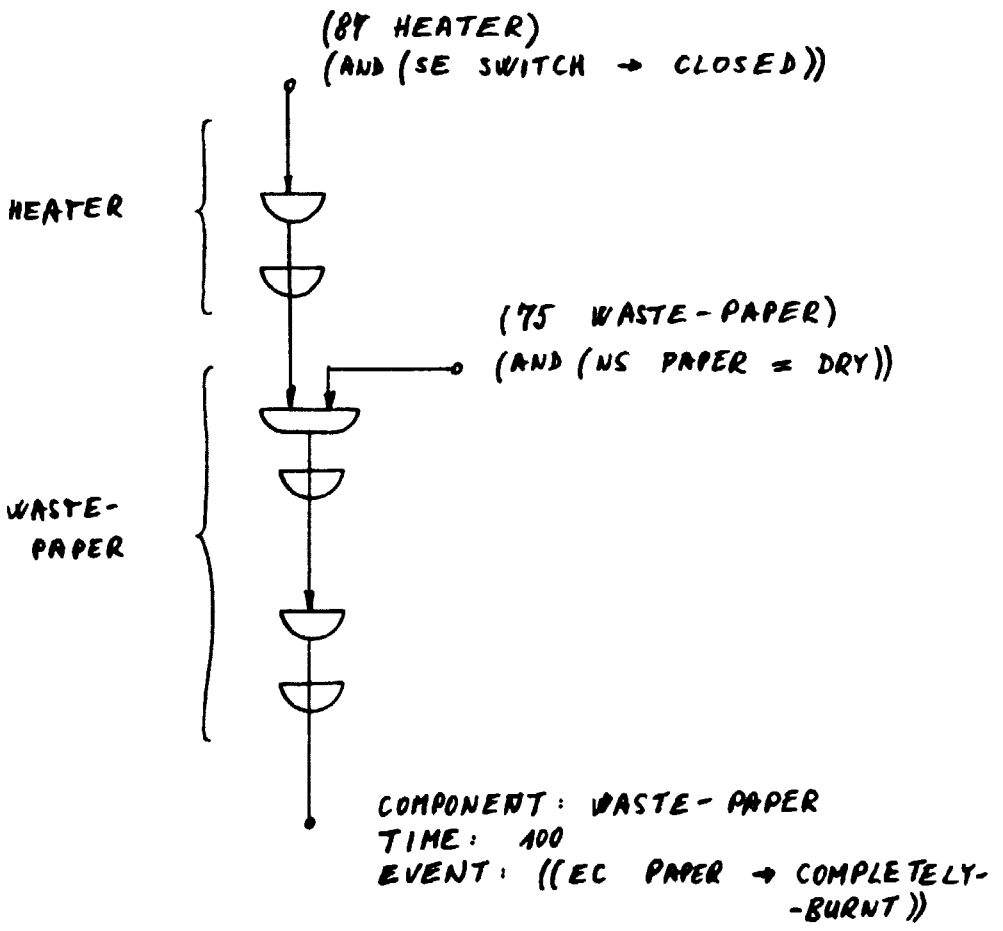


Fig. 8. Fault tree of COMPLETELY-BURNT WASTE-PAPER.

4. Present status and further developments

The described programs were implemented on PDP8 computer with 8k. The current limited storage capabilities have so far prevented application on real complex physical systems, therefore the early experiences are restricted into smaller examples. To get sufficient experiences with complex systems the implementation of programs on Burroughs 6700 computer are proceding, using a LISP8/FAURTRAN-IV Compiler/Interpreter. This implementation makes the connection of two algorithms possible and the consistency checking complete.

Another branch of current research work is the application of the fault tree and consequence diagram construction programs for plant disturbance analysis. As the sequential fault tree analysis can establish the possible logical combinations of primary faults for a distrubed plant situation and the consequence analysis can establish how far the disturbances extend, their combined application during distrubance analysis can effectively support the operator's work.

References

(1) - D.F. Haasl: Advanced concepts in fault tree analysis.
Paper presented at System Safety Sumposium, Seattle.
The Boeing Company, 1965.

(2) - D.S. Nielsen: The cause/consequence diagram method as a basis for quantitative accident analysis.
Risø-M-1374, May 1971.

(3) - J.B. Fussell, G.J. Powers, R.G. Bennets: Fault trees - a state of the art discussion.
IEEE Trans. on Reliability, Vol. 23/No. 1, 1974.

(4) - D.S. Nielsen: Use of cause-consequence charts in practical system analysis.
Paper presented at Reliability and Fault Tree Analysis Conf., Berkeley, September 1974.

(5) - J.B. Fussell: Synthetic tree model - a formal methodology for fault tree construction.
ANCR-1098, March 1973.

(6) - G.J. Powers, F.C. Tompkins: Fault tree synthesis for chemical processes.
AICHE Journal, Vol. 20/No. 2, March 1974.

(7) - S.A. Lapp, G.J. Powers: Computer-aided fault tree synthesis.
To be presented in IEEE Trans. on Reliability.

(8) - J.R. Taylor: A formalisation of failure mode analysis of control systems.
Risø-M-1654, September 1973.

(9) - J.R. Taylor: A semiautomatic method for qualitative failure mode analysis.
Paper presented at CSNI specialist meeting, Liverpool, April 1974.

(10) - J.R. Taylor: A language for heuristic programming.
To be published.

Appendix 1. Consequence diagram construction

Table 5. Detailed algorithm for consequence diagram construction

Start with data-base creation using component description list CL.

INPUT, Get initial event description of next initial component.
If no more initial components, go to CSELECT.
If initial component is not in CL, exit.
Place initial event on future event list of initial component.
Place initial component on active component list ACL marked with event time.
Go to INPUT.

CSELECT, Select main component MC with earliest event time ET from ACL and place it on SC.
If there are no more components on ACL, exit.
Delete SC from ACL.

Find the earliest event description on MC's future event list and place it on EV.

Find which of the connected following components are affected by EV.
Place affected components on the current following component list FL.

EVDEDUCE, Get next following component NC from FL.
If there are no components on FL, go to CSELECT.

Update the input variable values of NC using EV.
Deduce NC's current output event COE using input variable values, state variable values and transfer function.
If there is no output event, go to EVDEDUCE.

Table 5 cont.

Split COE into an immediate event IE and a delayed event DE.
If IE exists, update the state variable values of FC using their modified values involved in IE.

Add both IE and DE to NC's future event list.

Delete NC marked with old time values from ACL.
Evaluate new event time values using old time ET and time delay values of IE and DE.
Place NC on ACL, marked with new time values for IE and DE.
Go to EVDEDUCE.

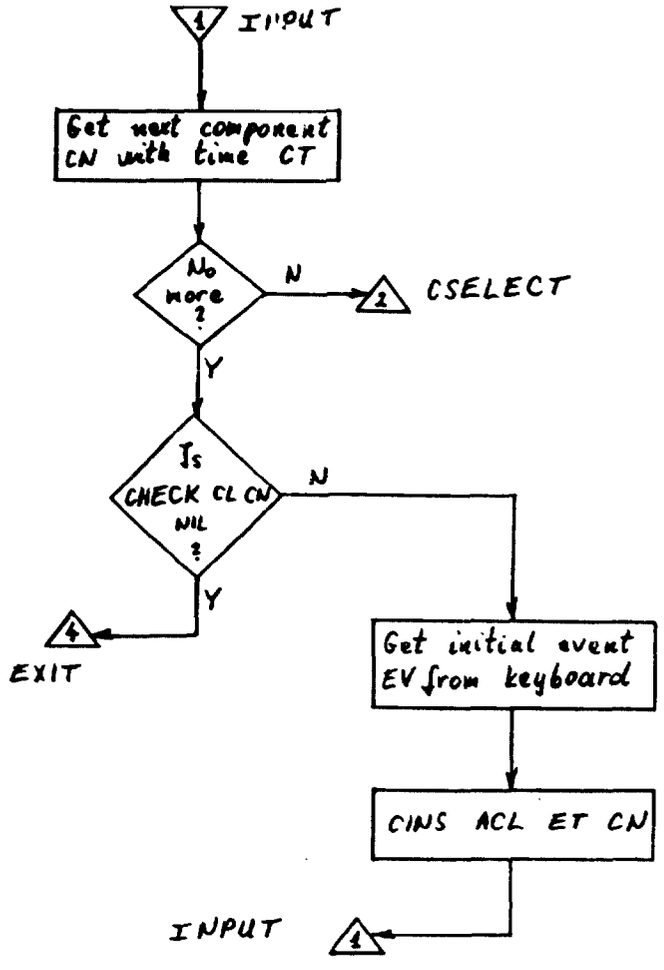


Fig. 9 Data input code.

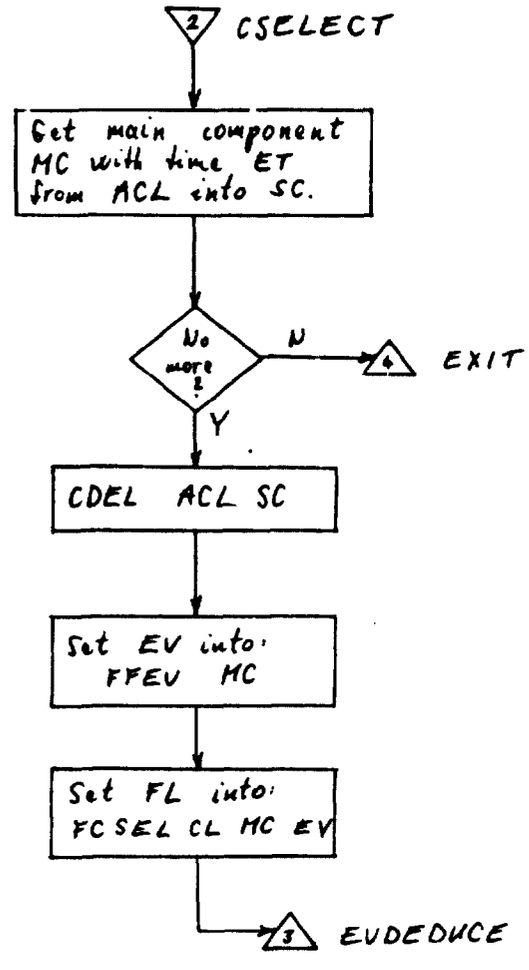


Fig. 10 Component selecting code.

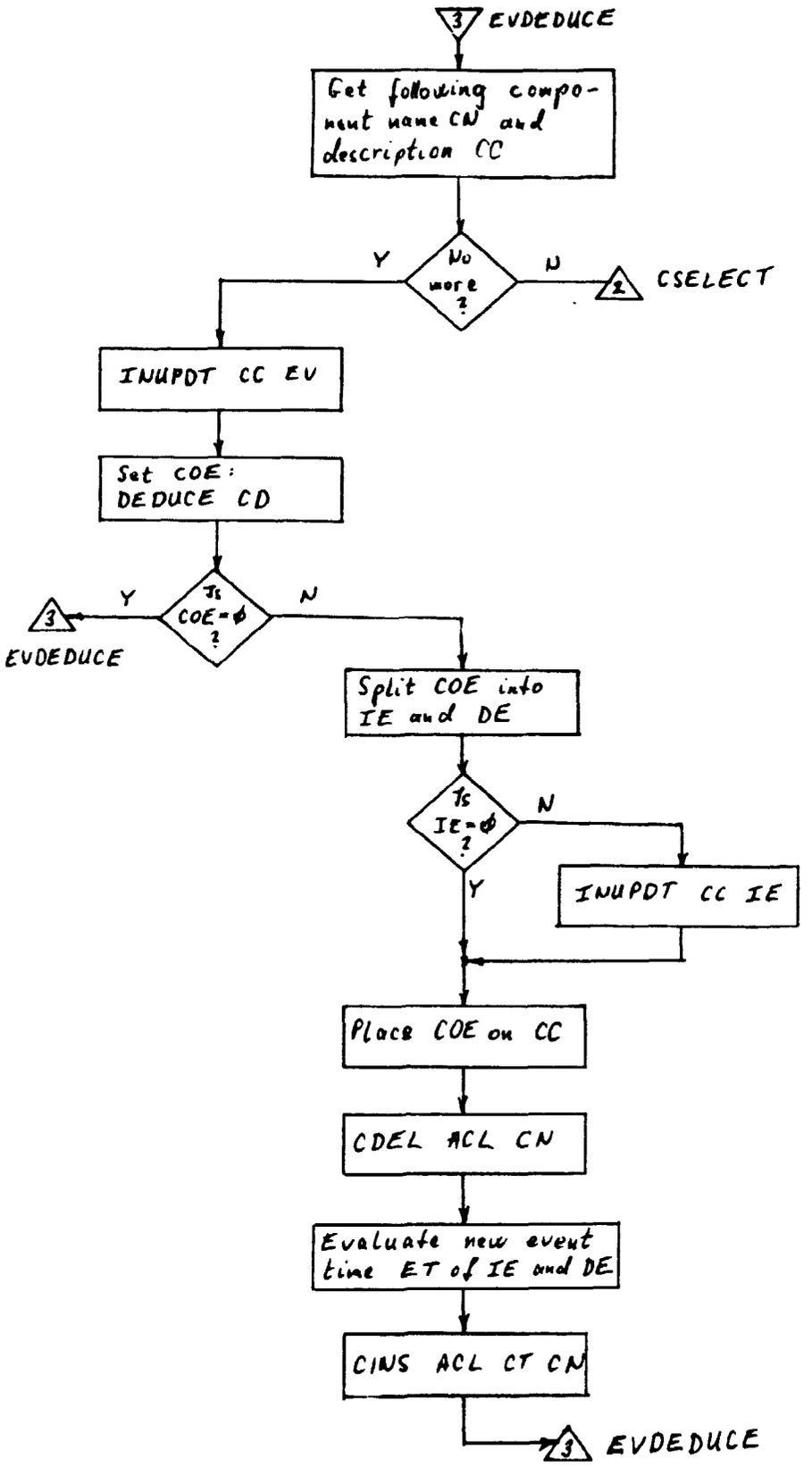


Fig. 11 Event deducing code.

Table 6. Structure of internal global variables

Set of active components	→ ACL
Component description list	→ CL
Following component list	→ FL
Component description	→ CD
Main component description	→ MC
Component name	→ CN
Output event with time	→ OE
Output event without time	→ EV
Current output event with time	→ COE
Immediate output event	→ IE
Delayed output event	→ DE
Event time	→ ET
Current event time	→ CT

```
ACL = ((⟨Time 1⟩ ⟨ACN1⟩ ⟨ACN2⟩...)
       (⟨Time n⟩ ⟨ACNi⟩ ⟨ACNi+1⟩...))
CL  = (⟨Comp. description 1⟩⟨Comp. description 2⟩...)
FL  = (⟨FCN1⟩ ⟨FCN2⟩....)
CD  = ⟨Comp. description⟩
MC  = ⟨Main comp. description⟩
CN  = ⟨Comp. name⟩
OE  = (⟨Time delay⟩ ⟨Mark⟩ ⟨EC1⟩⟨EC2⟩...)
EV  = (⟨EC1⟩⟨EC2⟩...)
COE = ((Ø ⟨Mark⟩ ⟨EC1⟩⟨EC2⟩...)
       (⟨Time delay⟩ ⟨Mark⟩ ⟨EC1⟩⟨EC2⟩...))
IE  = (Ø ⟨Mark⟩ ⟨EC1⟩⟨EC2⟩...)
DE  = (⟨Time delay⟩ ⟨Mark⟩ ⟨EC1⟩⟨EC2⟩...)
ET  = ⟨Abs. time⟩
CT  = ⟨Abs. time⟩
```

Appendix 2. Fault tree construction

Table 7. Detailed algorithm for fault tree construction

Start with data-base creation using component description list CL.
Get initial event description of initial component.
If initial component is not in CL, exit.
Place initial event marked with component name on selected output event list SO.
Create initial tree-branch, place it on oldbranch OBR.
Get OBR's pointer and place it on OPR.
Go to ANDBRANCH.

BRANCH, Get description CD of main component MC marked in OBR.
Get possible conjunctive combinations of its output variables and place them on IC.

ESEARCH, Get next conjunctive combination of input events/conditions from IC.
If there are no more combinations, go to BRMAKE.
Distribute the selected input events/conditions among the connected previous components of MC.
If one of them cannot be found in previous component's variable list, exit.
Place selected output events of previous components on SO and spontaneous event/normal state of MC on SI.

ANDBRANCH, Get next selected output event from SO.
If there are no more, create a new conjunctive sub-branch (AND-branch) and go to ESEARCH.
Search for possible respective input event/condition sets.
If there are not, go to ESEARCH.
Evaluate new event times for founded event/condition sets using absolute time value of OBR and time delays of SO.

Table 7 cont.

Create a new disjunctive sub-branch (OR-branch) using new input event/condition sets and their time.
Go to ANDBRANCH.

BRMAKE, Get new conjunctive sub-branch set.
If there is not, prune fault tree and go to NEXBRANCH.
Create a new tree-branch consisting of a printer PR and a logical function of events/conditions.

Create a new pointer set for next sub-branches using the created new tree-branch.
If there is no new pointers, build fault tree and go to NEXBRANCH.

Append new tree-branch into branch stack BS.
Append new pointer set into pointer stack PS.

NEXBRANCH, Get next pointer OPR of pointer stack.
If it is not null, get its relating branch from BS and place it on OBR, go to BRANCH.
Build and display final fault tree.
Exit.

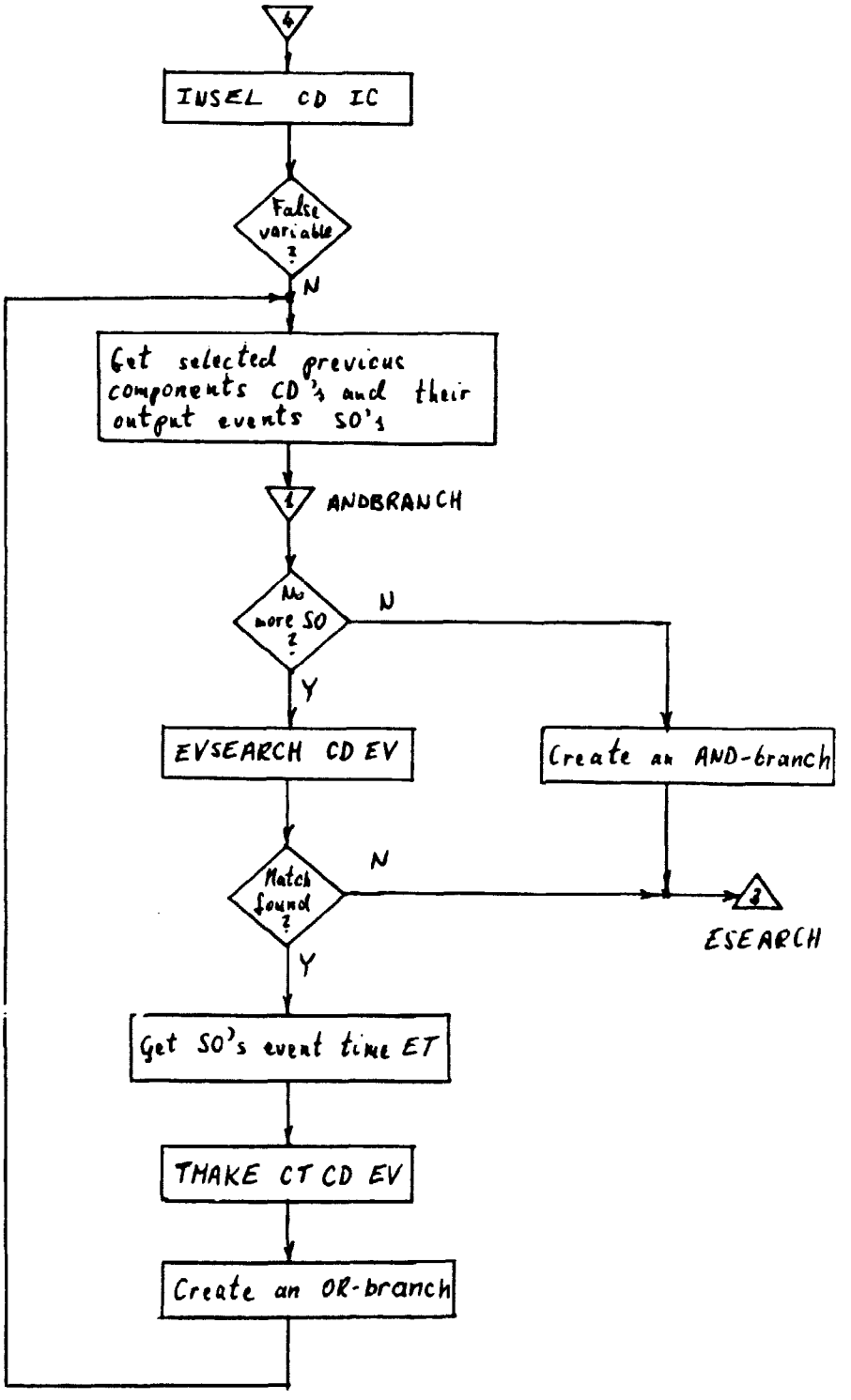


Fig. 12 Cause-event searching code.

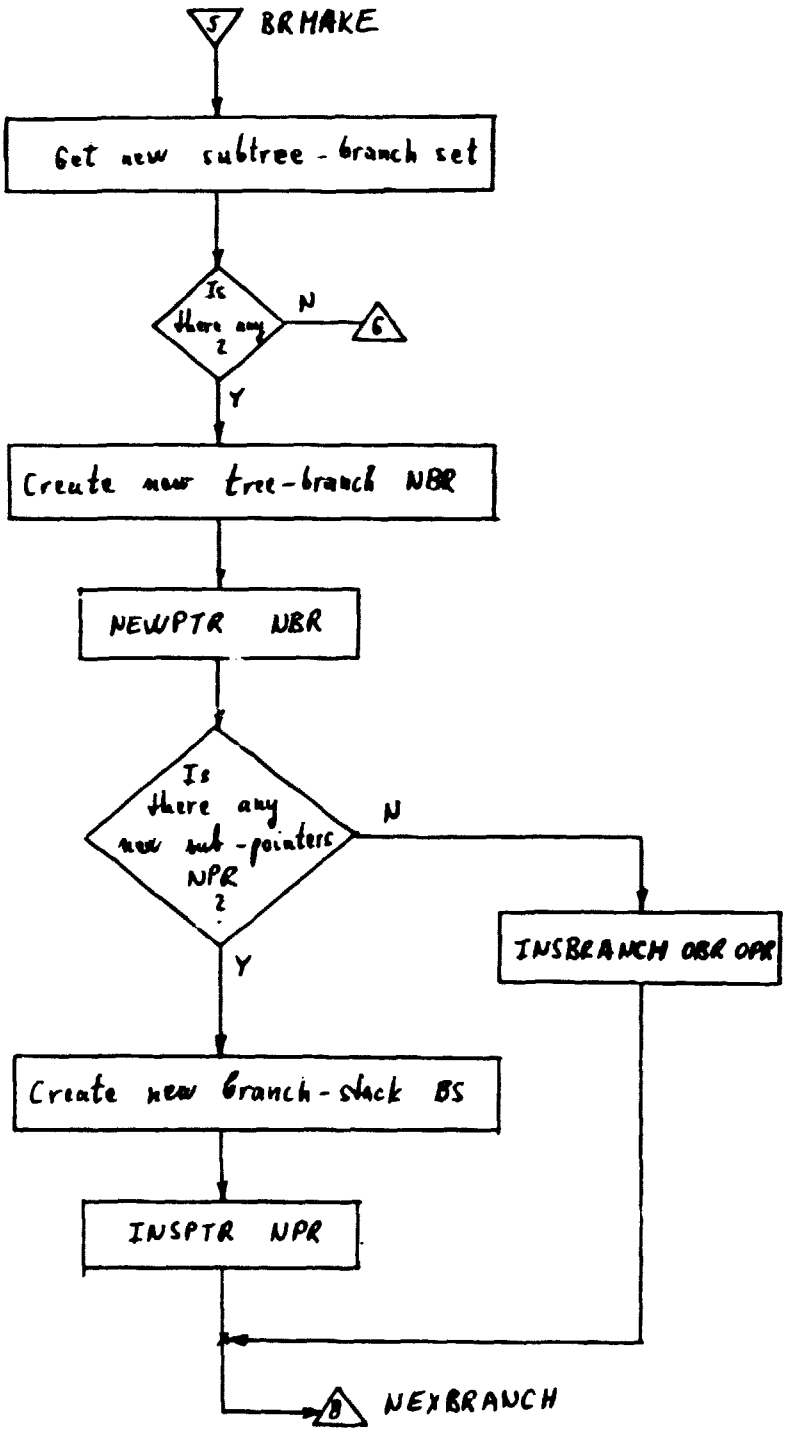


Fig. 13 Newbranch making code.

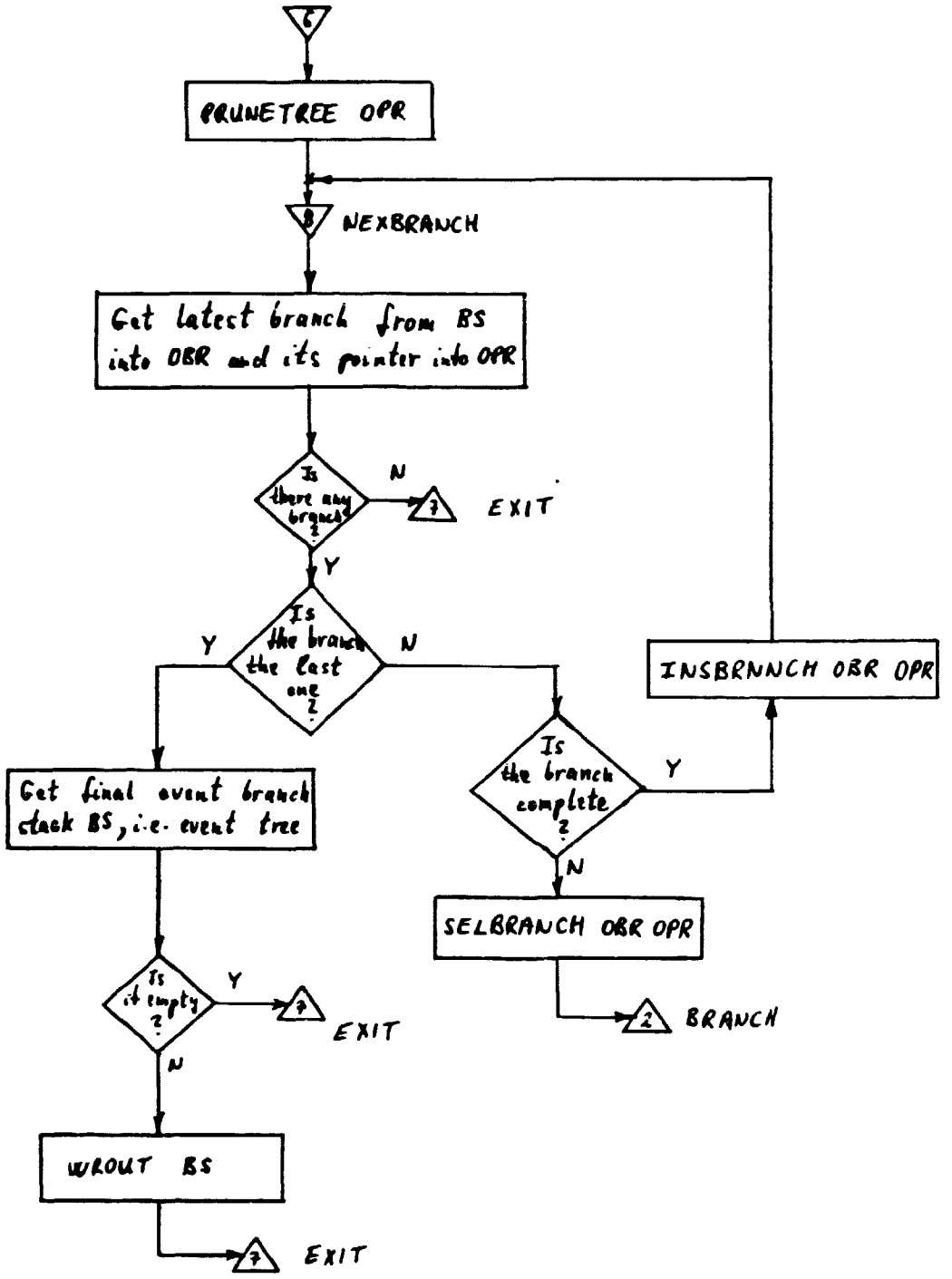


Fig. 14 Event tree making code.

Table 8. Structure of global variables

Initial component name	⟶	COMP
Initial output event	⟶	EVENT
Initial output event time	⟶	TIME
Component description list	⟶	CL
Component description	⟶	CD
Component name	⟶	CN
Distributed input variables	⟶	IV
Selected output events	⟶	SO
Selected spontaneous event/normal state	⟶	SI
Evaluated input event/condition sets	⟶	IS
Input event/condition combinations	⟶	IC
New tree-branch	⟶	NBR
Old tree-branch	⟶	OBR
Pointer of new tree-branch	⟶	NPR
Pointer of old tree-branch	⟶	OPR
Branch stack	⟶	BS
Pointer stack	⟶	PS
Event time	⟶	ET

```
COMP   =  ⟨Component name⟩
EVENT  =  (⟨EC1⟩⟨EC2⟩....)
TIME   =  ⟨Abs. time⟩
CL     =  (⟨Component description⟩⟨Component description⟩...)
CD     =  ⟨Component description⟩
CN     =  ⟨Component name⟩
IV     =  ((⟨MCName⟩ ⟨SE1⟩⟨SE2⟩...⟨NS1⟩⟨NS2⟩...)
           (⟨MCName⟩ ⟨EC1⟩⟨EC2⟩...              )
           (⟨PCN1⟩ ⟨EC1⟩⟨EC2⟩...                )
            :
           (⟨PCNn⟩ ⟨EC1⟩⟨EC2⟩...                ))
SO     =  ((⟨MCName⟩ ⟨EC1⟩⟨EC2⟩....)
           (⟨PCN1⟩ ⟨EC1⟩⟨EC2⟩...   )
            :
           (⟨PCNn⟩ ⟨EC1⟩⟨EC2⟩...   ))
SI     =  (⟨MCName⟩ ⟨SE1⟩⟨SE2⟩...⟨NS1⟩⟨NS2⟩...)
```

Table 8 cont.

```
IS   = ((<Time> (AND<VR1><VR2>...)(AND<VR1><VR2>...).....)
       (<Time> (AND...          )(AND...          ).....)
       :
       (<Time> (AND...          )(AND...          ).....))
IC   = ((AND<VR1><VR2>....       )(AND<VR1><VR2>....)....)
NBR  = ((<A.Time> <MCName>)
       (AND((<A.Time> <MCName>)(AND<SE1><SE2>...<NS1><NS2>....)
            (OR((<A.Time> <PCName>)(AND<VR1><VR2>....)(AND<VR1>...)...)
                 :
              (                                                    ))
              :
            (OR                                                    ))
         :
       (AND                                                      ))
OBR  = as NBR
NPR  = ((<A.Time> <MCName>)((<A.Time> <PCName>)(<A.Time> <PCName>).....))
OPR  = (<A.Time> <PCName>)
BS   = (<OBR1><OBR2>.....)
PS   = (<OPR1><OPR2>....<NPR1><OPRi><OPRi+1>...<NPR2>.....<NPRn>)
ET   = <Abs.Time>
```

Appendix 3. Outline of a LISP-8 code

```
/
* VARIABLES
⟨VN1⟩,Ø                          }
  :                              }   Variable names
                                 }

* FUNCTIONS
⟨ Subroutine name 1⟩             }
  :                              }   Subroutine names
 ⟨Subroutine name j⟩             }
Ø

EVAL
BEGIN GO START

/CONSTANTS
CRLF,CRLFPT                      }   Constants
  :                              }

*.+1&7776
CRLFPT,41ØØ;Ø                    }   Text constants
  :                              }

START,                           }
  :                              }   Main program
                                 }
END

⟨Subroutine 1⟩                   }
  :                              }   Subroutines
⟨Subroutine j⟩                   }

END
FREE,
$
```